Olioperustaisen ohjelmistomittauksen nykytila

Juha Gustafsson ja Lilli Nenonen

Helsingissä 1. marraskuuta 1999

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos
MAISA-projekti

Tiivistelmä

Ohjelmistojen mittaaminen on tapa varmistaa ja kehittää ohjelmiston laatua, ja sitä on harjoitettu lähes yhtä kauan kuin ohjelmointiakin. Siitä huolimatta erityisesti ohjelmistojen laatuun vaikuttavien tekijöiden mittaaminen on yhä haasteellinen ongelma ohjelmistotuotannossa. Ohjelmakoodi on yleisimmin käytetty mittauksen kohde, koska se on formaali esitystapa, helposti käsiteltävässä tekstuaalisessa muodossa ja luonnollisesti saatavilla. Myös olio-ohjelmien formaaleja arkkitehtuurikuvauksia voidaan mitata, jolloin mittaaminen voidaan tehdä jo ohjelmiston suunnitteluvaiheessa. Tämä olisikin edullisempaa kuin ohjelmakoodin analysointi, sillä mahdollinen muutosten tekeminen vaatii suunnitteluvaiheessa huomattavasti vähemmän kustannuksia kuin myöhemmin.
Osa hyvää olio-ohjelmistojen suunnittelua ovat suunnittelumallit. Niitä voidaan tunnistaa automaattisesti ohjelmiston arkkitehtuurista. Kun ohjelmasta tunnistetaan tietty suunnittelumalli, voidaan olettaa ohjelmistolla olevan kyseiselle mallille tyypillisiä ominaisuuksia. Tuore näkökulma ohjelmistojen mittaamiseen on yhdistää suunnittelumallien tunnistus ohjelmiston elinkaaren varhaiseen vaiheeseen.
Ohjelmistomittaukseen on kehitetty useita eri metriikoita. Perinteiset mittausmenetelmät sopivat ainakin suurelta osin myös olioperustaisten ohjelmistoarkkitehtuurien mittaamiseen. Olio-ohjelmien mittausta on tutkittu läpi 1990-luvun, ja on kehitetty erityisesti olioperustaisia metriikoita. Suunnittelumallien tunnistaminen laajentaa tavanomaisten metriikoiden antamaa kuvaa mitattavaan ohjelmistoarkkitehtuuriin.
Mittaustutkimuksen yhteydessä on kehitetty mittausohjelmistoja, jotka ovat oleellinen osa käytännön ohjelmistomittausta. Työkalut eivät useinkaan toimi täysin automaattisesti, vaan hyötyvät käyttäjän antamasta lisätiedoista ja ohjauksesta.

MAISA-projektin yhteystiedot

Jukka Paakki:	email Jukka.Paakki@cs.helsinki.fi
Inkeri Verkamo:	email Inkeri.Verkamo@cs.helsinki.fi
Juha Gustafsson:	email Juha.Gustafsson@cs.helsinki.fi
Lilli Nenonen:	email Lilli.Nenonen@cs.helsinki.fi

TEKES

Pauli Berg: email pauli.berg@tekes.fi

Kone

Salla Mannerjoki: email salla.mannerjoki@kone.com

Nokia Mobile Phones

Markku Ilvesmäki: email Markku.Ilvesmaki@nmp.nokia.com

Nokia Research Center

Anssi Karhinen: email Anssi.Karhinen@nokia.com

Space Systems Finland

Niklas Holsti: email Niklas.Holsti@ssf.fi

Sisältö

1 Johdanto

Ohjelmistojen mittaamista on harjoitettu lähes yhtä kauan kuin ohjelmointiakin. Siitä huolimatta erityisesti ohjelmistojen laatuun vaikuttavien tekijöiden mittaaminen on yhä haasteellinen ongelma ohjelmistotuotannossa. “Mitä ei voi mitata, sitä ei voi hallita” [Fen1991]. On suorastaan esitetty ohjelmistotuotannon olevan kriisivaiheessa, koska ohjelmia tuotetaan yhä enenevässä määrin, ilman että ohjelmistoja kuitenkaan osataan mitata siinä määrin kuin teollisesti tuotettavaa tuotetta pitäisi. On yleisesti tunnustettu, että laadukas ohjelmistoprosessi tuottaa laadukkaita ohjelmia. Tämä ei kuitenkaan riitä, vaan on määriteltävä, millainen on laadukas ohjelma. Vasta tällöin on mahdollista tietää, ovatko prosessin aikana syntyneen ohjelman laatuominaisuudet sellaiset kuin haluttiin. Teollinen tuotanto ei saa vaikuttaa taikurin toiminnalta: otetaan hattu, tempaistaan siitä tuote ja katsotaan, millainen siitä tuli. Täytyy tietää, mikä on laadukasta, jotta voidaan tuottaa laatua. Ja jotta voidaan tietää, onko ohjelmisto laadukas, sen ominaisuuksia on pystyttävä mittaamaan.

Ohjelmakoodi on yleisimmin käytetty mittauksen kohde, koska se on formaali esitystapa, helposti käsiteltävässä tekstuaalisessa muodossa ja luonnollisesti saatavilla. Mittaaminen olisi kuitenkin edullisempaa tehdä jo ohjelmiston suunnitteluvaiheessa, jolloin muutosten tekeminen vaatii huomattavasti vähemmän kustannuksia kuin siinä vaiheessa, jolloin ohjelmakoodi on jo valmis [Pre1997]. Ohjelmistoarkkitehtuurin mittaaminen ohjelmiston suunnitteluvaiheessa vaatii luonnollisesti arkkitehtuurin formaalia kuvausta. Ohjelmoinnin olioparadigma on tuonut mukanaan tällaisia kuvaustekniikoita, jotka (paradigman mukaisesti) kulkevat mukana läpi ohjelmiston koko elinkaaren. Oliopohjainen ohjelmistojen staattisen ja dynaamisen rakenteen monipuolinen kuvauskieli, UML, on eräs sopiva vaihtoehto mitattavan ohjelman esitystavaksi [UML].

Osa hyvää olio-ohjelmistojen suunnittelua ovat *suunnittelumallit* [Gam1995]. Ne ovat hyväksi havaittuja keinoja ratkaista tietty, yleinen oliosuunnitteluongelma. Suunnittelumallien käyttö lisää ohjelman uudelleenkäytettävyyttä, laajennettavuutta ja siten myös ylläpidettävyyttä [Bec1996, Sch1995]. Hyvän ratkaisun vastapainoksi kullakin suunnittelumallilla on huonotkin ominaisuutensa, minkä vuoksi yksi malli ei sovi saman ongelman kaikkiin ratkaisutilanteisiin. Tiettyä suunnittelumallia käyttävän ohjelman rakenteella on vastaavasti kyseiselle mallille tyypillisiä hyviä ja huonoja ominaisuuksia [Gam1995, Men1996] Suunnittelumalleja on mahdollista tunnistaa ohjelmakoodista ja arkkitehtuurikuvauksista edellyttäen, että niiden rakenne on formaalisti dokumentoitu [Mis1997]. Kun ohjelmasta tunnistetaan tietty suunnittelumalli, voidaan olettaa sen rakenteella olevan mallin mukanaan tuomia ominaisuuksia. Suunnittelumallien lisäksi on olemassa ns. *negatiivisia(anti-)suunnittelumalleja* (anti-patterns), jotka ovat huonoiksi havaittuja ratkaisuja tiettyyn ongelmaan, ja siten suunnittelumallien vastakohtia [CMG, Bro1998]. Niitäkin voidaan tunnistaa ohjelman rakenteesta. Mallien tunnistaminen on perinteisesti liittynyt ohjelmistotuotannon *käänteistekniikkaan* (reverse engineering), jonka keinoin pyritään ymmärtämään olemassa olevaa ohjelmaa, jonka dokumentaatio on puutteellinen tai vanhentunut. Suunnittelumallien tunnistaminen auttaa ymmärtämään ohjelman suunnitteluratkaisuja ja toimintaa.

Tuore näkökulma ohjelmistojen mittaamiseen on yhdistää suunnittelumallien tunnistus ohjelmiston elinkaaren varhaiseen vaiheeseen. Jos ohjelman suunnitteluvaiheessa tehdystä arkkitehtuurikuvauksesta löydetään malleja, voidaan ohjelman tulevia ominaisuuksia päätellä löydettyjen mallien ominaisuuksien avulla. Havaittujen anti-suunnittelumallien mukanaan tuomat laatuongelmat voidaan korjata mahdollisimman varhaisessa vaiheessa. Suunnittelumalleista saatavan informaation yhdistäminen perinteisiin metriikoihin antaa lisäksi laajemman kuvan ohjelman ominaisuuksista. Käänteistekniikan työkalujen käyttäminen jo suunnitteluvaiheessa tavallaan yhdistää kaksi asiaa, jotka teoriassa ovat ohjelmistoprosessin eri päissä. Idea toimii, koska mittauksen kohde – arkkitehtuurikuvaus – on kummassakin päässä sama.

Seuraavassa luvussa käsitellään ohjelmistojen ja arkkitehtuurien mittausta suhteessa muuhun laadunvarmistustoimintaan ohjelmistoprosessissa. Luvussa 3 luodaan katsaus ohjelmistomittauksen teoriaan ja käsitteisiin. Luvussa 4 kerrotaan oliopohjaisista ohgjelmistometriikoista ja niiden tutkimuksesta. Luvussa 5 pohditaan avoimia kysymyksiä sekä UML:a mittauskohteena, sen ongelmia ja erästä UML:n vaihtoehtoa. Luvussa 6 tehdään yhteenveto ohjelmistomittauksen nykytilasta.

2 Mittaaminen ohjelmistoprosessissa

Ohjelmistoprosessiin liittyy keskeisesti toisaalta prosessin ja toisaalta tuotettavan ohjelmiston laatu. Ohjelmiston laatu eli ohjelman ja käyttäjän vaatimusten ja odotusten vastaavuus on jo määritelmänsä mukaan subjektiivinen ja verraten abstrakti käsite. Osa kiinteästi laatuun sidoksissa olevista asioista on kuitenkin hyvin selkeästi määriteltävissä ja mitattavissa. Tällaisia asioita ovat esimerkiksi luokkien välisten kytkentöjen määrä sekä monet suorituskykyyn liittyvät seikat.

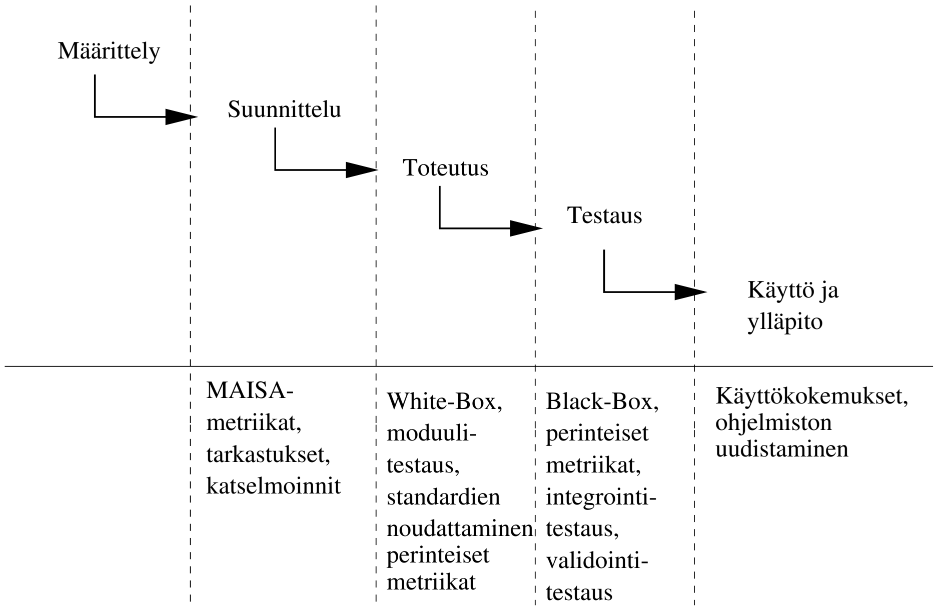
2.1 Mittaaminen laadunvarmistuksessa

Ohjelmiston laadunvarmistus (software quality assurance, SQA) on koko ohjelmistoprosessin käsittävä mekanismi, joka sisältää mm. kuvassa 1 esitetyt tekniikat [Hum1990, ISO, Pre1997]. Menettelyllä pyritään varmistumaan paitsi yksittäisten osatuotosten (dokumentti, ohjelmamoduuli) tuloksista, myös luomaan edellytykset korkeatasoiselle ohjelmistoprosessille. Itse prosessin ja samalla tuotettavien ohjelmien laadun halutaan siis ajan myötä paranevan. Kuvassa 1 on esitetty laadunvarmistuksen apuvälineistöä ohjelmistoprosessin eri vaiheissa (esimerkkinä vesiputousmalli).

Ohjelmistoprosessia itseään voidaan mitata mm. seuraavilla mittareilla:

- ohjelmarivien määrä henkilötyökuukaudessa
- tehtävään käytetty aika henkilötyökuukaudessa
- aikataulun pitävyys (prosentteina)
- valmistajan havaitsemien virheiden määrä
- loppukäyttäjien raportoimien virheiden määrä.

Kattava ohjelmistoprosessin laadun arviointiin kehitetty tekniikka on projektin kypsyysmalli (Capability Maturity Model, CMM [Hum1990, Hum1995, Pau1993]). Mallissa



Kuva 1: Ohjelmistoprosessi ja laadunvarmistus

arvioidaan organisaation ohjelmistoprosessissa käytettyjä menetelmiä ja periaatteita. Arvioinnin perusteella prosessi luokitellaan johonkin CMM:n viidestä kypsyystasosta. Samantapaisia tekniikoita pienemmässä mittakaavassa ovat ryhmän ohjelmistoprosessi (Team Software Process, TSP) sekä edelleen henkilökohtainen ohjelmistoprosessi (Personal Software Process, PSP). Ohjelmistometriikat tukevat tällaista prosessin arvointia mm. tuottamalla konkreettisia lukuarvoja, joiden perusteella prosessin laatua voidaan arvioida.

Ohjelmatasolla laadunvarmistuksen tavoitteita ovat suoranaisten virheiden poistaminen sekä potentiaalisten ongelmakohtien paikallistaminen. Keskeisiä asioita ovat esimerkiksi hyvien ja huonojen suunnitteluratkaisujen ja suorituskyvyn kannalta kriittisten kohtien löytäminen sekä ohjelman helppokäyttöisyys. Mittaaminen mahdollistaa konkreettisten, verrattavissa olevien arvojen hankkimisen. Mittaamisen aiheuttaman rasitteen (ml. lisätyö ja kustannukset) on oltava suhteessa saavutettuun hyötyyn. Vastapainona voidaan ajatella olevan mm. kielteisen palautteen, myöhemmin tehtäväksi tulevien päivitysten ja takuukorjausten.

Kaikkien ohjelmistoprosessin vaiheiden tulisi tuottaa ennalta kiinnitetyt laatukriteerit täyttäviä tuloksia. Arkkitehtuurimittarit muodostavat taustan, jota vasten näitä kriteereitä voidaan tarkastella. Yleisiä laatutekijöitä ovat mm. seuraavat (katso myös luku 3.2):

- *Oikeellisuus* (correctness) tarkoittaa sitä, että ohjelma toimii vaatimusten mukaisesti. Tavanomaisin oikeellisuusmittari on virhetoimintojen määrä suhteessa ohjelmarivien määrään.
- *Ylläpidettävyyteen* (maintainability) liittyy läheisesti mm. seuraavat seikat:
  - virheiden korjauksen helppous (maintainability)
  - ohjelman täydentäminen ja laajentaminen (flexibility) ja
  - ohjelman muuntaminen muihin ympäristöihin (portability).

  Ylläpidettävyydelle käyttökelpoisia mittareita ovat mm. kiinteys (kukin komponentti on selkeästi yhteenkuuluva kokonaisuus) sekä kytkentä (komponenttien välisten yhteyksien lukumäärä).
- *Käytettävyys* (usability) tarkoittaa sitä, kuinka käyttäjäystävällinen ohjelmisto on. Tämän mittaaminen on runsaasti tutkittu aihealue, joskin objektiivisten mittarien kehittäminen on hankalaa.
- *Tehokkuudella* (efficiency) tarkoitetaan sitä, kuinka nopeasti ja millaisella muistimäärällä ohjelma suorittaa sille annetut tehtävät.Tämä on tärkeä ominaisuus erityisesti tosiaikaisissa ja sulautetuissa järjestelmissä. Tehokkuutta on onneksi usein kohtuullisen helppoa mitata.

2.2 Mittaamisen suhde muihin laadunvarmistusmenetelmiin

2.2.1 Testaus

Testaus on vakiintunut ja huomattavan laaja osa ohjelmistoprosessia [Hum1990, Mye1979, Pre1997]. Testauksen käytännön vaikeuksista saa kuitenkin hyvän kuvan käyttämällä lähes mitä tahansa kaupallista ohjelmistoa: virheisiin törmää huolestuttavan usein.

Rakenteellisessa testauksessa (white-box testing) pyritään selvittämään ohjelmaan kirjoitettujen lauseiden sisäisen toiminnan oikeellisuutta. Testausta voidaan tehdä heti kun yksittäinen moduuli on saatu valmiiksi.

Toiminnallisessa testauksessa (black-box testing) pyritään varmistumaan ulkoisen ohjelmalogiikan toiminnasta ottamatta kantaa ohjelman toteutuksen yksityiskohtiin [Bei1995].

Viimeinen vaihe testausprosessia ovat integrointi- ja validointitestaus. Integrointitestauksessa selvitetään koko ohjelmiston osien keskinäistä yhteistoimintaa. Kuten ihmisten välisessä yhteistoiminnassakin, tämä saattaa osoittautua hankalaksi. Validointitestauksessa puolestaan vastataan kaikkein vaikeimpaan kysymykseen - tekeekö ohjelma sitä, mitä sen haluttiin tekevän.

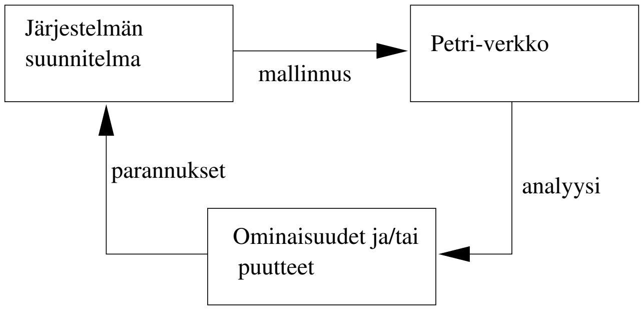
Edellä mainittuihin testausvaiheisiin on kehitetty lukuisa joukko yksityiskohtaisia menetelmiä, joilla useimmat suoranaiset virheet on mahdollista löytää. Kuitenkin kahteen perusongelmaan on vaikea päästä käsiksi pelkästään näillä tavoin. Ensinnäkin ohjelma on jo lähes valmis testausvaiheessa, joten korjausten teko voi olla hankalaa tai kallista. Toiseksi monet ongelmat ovat liian syvällä ohjelman, menetelmien tai itse ohjelmistoprosessin rakenteessa, jotta ne voitaisiin havaita testaamalla. Suunnitteluvaiheen ohjelmistomittaus pyrkii vastaamaan ainakin osaan näistä ongelmista.

2.2.2 Formaalit menetelmät

Koska tietojenkäsittely on läheistä sukua matematiikalle, on luonnollista, että matemaattispohjaisia menetelmiä on pyritty soveltamaan ohjelmien arvioinnissa. Syntyneitä tekniikoita ovat mm. formaali spesifiointi ja verifiointi [Pot1996], Petri-verkot [Pet1981] sekä suorituskykyanalyysi [Men1994, SaC1981, Smi1990].

Formaalissa spesifioinnissa loogiset, mutta tietyllä tasolla kuitenkin epätäsmälliset ohjelmistolle asetetut vaatimukset mallinnetaan matemaattisessa muodossa, jolloin niitä voidaan arvioida matemaattisilla menetelmillä. Tavoitteena on ohjelman oikeellisuuden ja täydellisyyden systemaattinen varmentaminen eli verifiointi. Formaali spesifiointi ja verifiointi perustuvat joukko-operaatioihin ja predikaattilogiikkaan.

Petri-verkot kehitettiin jo 1960-luvun alussa, joten ne ovat hyvinkin varhainen tekniikka laadunvarmistukseen. Ne on suunniteltu formaaliksi mallinnusmenetelmäksi järjestelmiin, joissa on keskenään kommunikoivia, rinnakkaisia komponentteja. Formaalin spesifioinnin tapaan ne soveltuvat ohjelman toiminnan ja kompleksisuuden mallintamiseen, mutta taustana on nyt verkkoteoria (kuva 2).



Kuva 2: Petri-verkkojen käyttö

Tosiaikaisissa ja sulautetuissa järjestelmissä tärkein laatukriteeri on usein ohjelman suoritusaika: joko puhtaasti ohjelman nopeus tai ainakin suoritusajan ennustettavuus. Suorituskyvyn ennustamisen välttämättömyys on luonut oman tieteenalansa, suorituskykyanalyysin. Yksinkertaisimmillaan tämä perustuu jonoverkkomalleihin. Niissä järjestelmä tai sen osa kuvataan ja sen toimintaa (aika, kapasiteetti tms.) simuloidaan verkkoteorian keinoin.

Edellä lueteltujen menetelmien ideana on hyödyntää matemaattista välineistöä ja formaalia todistusvoimaa ohjelmistojen laadun varmistamisessa. Epämääräiset laatuvaatimukset pyritään saamaan kvantitatiiviseen, todistettavaan muotoon. Tässä mielessä ne ovat läheistä sukua mittaukselle.

2.2.3 Katselmukset ja tarkastukset

Ryhmätyönä suoritettavat katselmukset (reviews) ja tarkastukset (inspections) ajoittuvat usein suunnitteluvaiheeseen kuten suunnittelumalleja hyödyntävät ohjelmistomittausmenetelmätkin. Tavanomaisten ohjelmointivirheiden löytäminen tarkastuksilla on noin 2-4 kertaa nopeampaa kuin testaamalla [Rus1991]. Jopa 80% ohjelmakoodissa alun perin olleista virheistä on löydettävissä tarkastuksilla.

Tarkastusten ajatuksena on etsiä kohteesta virheitä systemaattisella läpikäynnillä - siis eräänlaisella aivoriihityöskentelyllä. Eriasteisia tarkastuksia voidaan tehdä kaikissa ohjelmistoprosessin vaiheissa. Itse tarkastusmenettelyn läpiviemiseen on kehitetty yksityiskohtaisia ohjeita [Pre1997].

Huonot suunnitteluratkaisut ja muut rakenteelliset ongelmat voivat löytyä hyvin tehdyssä tarkastuksessa, jolloin ollaan jo lähellä suunnitteluvaiheen mittauksille asetettuja tavoitteita. Valitettavasti tällaiset 'metatason' ongelmat voivat yhtä hyvin jäädä myös löytymättä. Jos projektiryhmä on keskenään ratkonut tiettyä suunnitteluongelmaa, voi saman ryhmän tekemässä tarkastuksessa jäädä metsä näkemättä puilta. Omille ratkaisuilleen tulee helposti sokeaksi.

Tarkastuksissa löydetyistä virheistä voidaan tietysti laskea lukumääriä ja vakavuusasteita, mutta kvantitatiivisia mittoja on silti vaikea keksiä. Suhde mittaukseen löytyy siis tästä lähestymistapaerosta: mittauksilla etsitään täsmällisiä lukuarvoja.

2.2.4 Standardit ja käytännöt

Johdonmukaisesti noudatettavalla laatustrategialla on merkittävä välillinen vaikutus ohjelmistotuotteen laatuun [Hum1990, ISO]. Vakiintuneiden merkintätapojen käyttö esimerkiksi ohjelmakoodia kirjoitettaessa voi edesauttaa virheiden välttämisessä, kun toisen henkilön tekemää koodia on helpompi lukea.

Edellä mainittuun ylläpidettävyyteen johdonmukaisella ohjelmistoprosessin standardoinnilla on erityisen suuri vaikutus. Virheiden korjaus ja uusien ominaisuuksien lisäys on ohjelmistoa entuudestaan tuntemattomallekin vaivattomampaa, jos merkintätavat tunnetaan. Olioperustaiset menetelmät tarjoavat tähän vielä oman lisänsä mm. perinnän ja kapseloinnin muodossa.

2.3 Suunnitteluvaiheessa tehdyn mittauksen erityispiirteitä

Suunnitteluvaiheessa tehdyn arkkitehtuurimittauksen ilmeinen etu liittyy luonnollisesti mittauksen ajankohtaan ohjelmistoprosessissa. Varhaisessa vaiheessa tehdyn mittauksen tuloksena löytyneet ongelmakohdat on helpompi ja halvempi korjata kuin jo lähes valmiin tuotteen kyseessä ollessa. Mitä suuremmasta rakenteellisesta ongelmasta on kyse, sitä oleellisempaa on löytää se aikaisessa vaiheessa.

Toinen merkittävä kysymys on, löydetäänkö rakenteellisia ongelmia lainkaan tavanomaisella testauksella. Mitä perustavampaa laatua nämä ovat, sitä huonommat ovat mahdollisuudet niiden löytämiseen. Lisäksi aiheutuneet haitat ovat tällöin suuremmat.

Onko järkevä mittaus sitten ylipäänsä mahdollista ilman ohjelmakoodia? Saadaanko suunnitteluvaiheen UML-kaavioista tms. irti tuloksia, jotka korreloivat valmiin tuotteen ominaisuuksien kanssa (jollain asteikolla mitattuna)? Suunnittelumalleista ja niiden ominaisuuksista saatujen kokemusten perusteella voidaan odottaa myönteistä vastausta.

Jos arkkitehtuurimittareita onnistutaan kehittämään, kuinka hyvä näin saatujen tulosten laatu on verrattuna muissa vaiheissa laskettuihin perinteisiin metriikoihin? Mittauksen perustana oleva materiaali (eli UML-kaaviot) on huomattavasti karkeammalla tasolla kuin tehtäessä mittausta myöhemmin, joten eroja tulee olemaan. Kyse on siitä, kuinka paljon epätarkkuus lähdemateriaalissa vaikuttaa tulosten laatuun. Selvästikin tässä vaiheessa tarvitaan interaktiota käyttäjän kanssa.

Kiintoisa kysymys on myös, minkälaisiin ongelmiin suunnitteluvaiheessa tehty analyysi sopii erityisen hyvin tai huonosti. Tähän vaikuttaa mm. se, ovatko kohdealueella tyypillisesti esiintyvät suunnittelumallit helposti löydettävissä.

3 Mittaaminen ja metriikat

Seuraavassa käsitellään ohjelmistomittauksen teoriaa ja siihen liittyviä käsitteitä pohjaksi ohjelmistomittauksen tutkimuksen nykytilan kartoittamiselle seuraavassa luvussa. Toisessa aliluvussa kerrotaan ohjelmiston laadun mittaamisesta.

3.1 Mittaamisen teoriaa

Mittaaminen on mittauksen kohteen kuvaamista numeeriselle arvoasteikolle. Jotta pystyttäisiin mittaamaan, on tiedettävä, mikä on mittauksen kohde ja mikä on sen mitattava ominaisuus eli suure. Mittauksen kohteeseen liittyy mittauksen *granulariteetti*, toisin sanoen mittauksen *entiteetti* (entity) [Fen1991]. Mitataanko tietyn metodin, luokan vai koko ohjelmiston ominaisuutta? Jos mitataan ohjelmistoa, mukaan tuskin lasketaan ohjelman mahdollisesti käyttämiä kirjastoja. Mitattavat entiteetit voidaan jaotella ohjelmistoprosessin eri vaiheiden mukaan tuotteisiin, prosesseihin ja resursseihin. Resurssit ovat ohjelmistoprosessin syötteitä, kuten henkilöstön määriä. Prosessit liittyvät tietyn tehtävän suorittamiseen tuotantoprojektissa, ja niistä mitataan esimerkiksi tiettyyn tehtävään kulutettuja miestyötunteja. Tässä keskitytään tarkastelemaan pääasiassa tuotteisiin liittyvää mittaamista. Tuotteisiin kuuluvat jokaisen tuotantoprosessin vaiheen tulokset: enemmän tai vähemmän formaalit dokumentit sekä ohjelmakoodi.

Entiteetin mitattavia ominaisuuksia sanotaan sen *attribuuteiksi*, ja kullakin entiteetin edustajalla sama attribuutti saa oman arvonsa. Osalle attribuuteista arvot ovat mitattavissa tai laskettavissa suoraviivaisesti entiteetun sisäisestä rakenteesta. Niitä sanotaan *sisäisiksi attribuuteiksi*. Toista osaa attribuuteista ei voi mitata suoraan, vaan niiden arvot saadaan laskemalla toisten attribuuttien avulla. Tällaisia sanotaan *ulkoisiksi attribuuteiksi* [Fen1991]. Esimerkiksi entiteetillä “luokka” on sisäiset attribuutit “metodien lukumäärä” ja “kenttien lukumäärä” sekä ulkoinen attribuutti “kiinteys” (cohesion). Tietyllä luokalla X attribuutin “metodien lukumäärä” arvo on vaikkapa viisi ja “kenttien lukumäärä”n arvo on kolme. “kiinteys”-attribuutin arvo lasketaan luokan metodien, kenttien ja assosiaatioiden ominaisuuksien avulla (tähän on useita eri tapoja). Koosteisen entiteetin (pakkaus) attribuuttien arvot on voitava laskea inkrementaalisesti pienempien entiteettien arvoista. Sisäisten attribuuttien arvojen laskemista sanotaan *suoraksi mittaamiseksi* ja ulkoisiin attribuutteihin perustuvaa mittaamista *epäsuoraksi mittaamiseksi*. Mittaamisen yhteydessä usein käytetty termi *metriikka* (metrics) tarkoittaa sisäisten attribuuttien avulla tapahtuvaa suoraa mittaamista, kun taas *mitta* (measure) tarkoittaa epäsuoraa mittaamista [Fen1991, FeP1996]. Seuraavassa "mittaamisella" tarkoitetaan yleisesti sekä suoraa että epäsuoraa mittaamista.

Mitattaville attribuuteille on oltava vähintään yksi numeerinen esitystapa eli kuvaus mitattavalta entiteetiltä numeeriselle mitta-asteikolle. Tällöin voidaan ajatella, että attribuutti *on* tämä kuvaus. Lisäksi kuvauksen on oltava induktio, toisin sanoen tietyn entiteetin tietyllä attribuutilla on vain yksi arvo tietyssä kuvauksessa. Mikäli attribuutilla on olemassa useita kuvauksia eli useita eri mitta-asteikoita, kuvaukset eivät saa olla ristiriidassa toistensa kanssa [Fen1991]. Kuvausten välillä on oltava funktio, joka muuttaa toisen kuvauksen asteikon toiselle asteikolle. Tätä sanotaan *skaalaamiseksi*. SEI:n (Software Engineering Institute) teknisessä raportissa ja Norman Fentonin mittausoppikirjassa asetetaan järkeville mittausjärjestelmille seuraavia kriteerejä edellämainittujen lisäksi. [ArS1995, Fen1991].

- **Mittausjärjestelmän on oltava vakaa.** Mittauksen on tuotettava sama tulos toistettuna samalle mittauksen kohteelle, riippumatta vähäisistä muutoksista ympäristössä ja mittaustyökalussa. Mittaamisen on oltava objektiivista.

- **Mittausjärjestelmän on sisällettävä mitta-asteikko ja rajat.** Mitattavia tuloksia on voitava verrata toisiinsa, toisin sanoen mitta-asteikon on sisällettävä vertailurelaatio. Lisäksi on tiedettävä, onko suurempi arvo suotuisampi tulos kuin pienempi. Mitta-asteikolla on oltava rajat.

- **Mittausjärjestelmän on oltava mielekäs.** Mitattavan suureen täytyy liittyä mittauksen kohteeseen, ja mitattavien arvojen on oltava laskettavissa mittauksen kohteena olevasta entiteetistä.

3.2 Mittaaminen ja laatu

Seuraavaan taulukkoon on listattu tyypillisiä ohjelmiston attribuutteja. Rakenteisella koolla tarkoitetaan metodien lukumäärää luokassa, luokkien lukumäärää ja muita vastaavia. Koolla ulkoisena attribuuttina tarkoitetaan ohjelmakoodin kokoa riveinä ja käännetyn ohjelman kokoa kilotavuina. On huomattava, että ne ominaisuudet, jotka yleisesti vaikuttavat ohjelmiston laatuun, ovat ulkoisia attribuutteja, ja siten niiden vaikeammin mitattavissa kuin yksinkertaisemmat ominaisuudet, jotka ovat ohjelmiston sisäisiä attribuutteja. Metriikat ovat rajalliset, niiden perusteella voidaan sitten tehdä päätelmiä laadusta yms.

Laatu itsessäänkin on ohjelmiston ulkoinen attribuutti, jonka mittaamisen määrittelee käytetty *laatujärjestelmä* [ISO]. On huomattava, että laadun määritelmä vaihtelee ohjelmasta riippuen. Esimerkiksi ohjelman koko saattaa olle hyvinkin vähämerkityksellinen laatukriteeri, kun taas modulaarisuus on tyypillisesti yksi tärkeimpiä laadun mittareita.

Sisäisiä attribuutteja (metriikoita)	**Ulkoisia attribuutteja** (mittoja)
Kytkentä (coupling)	Uudelleenkäytettävyys (reusability)
Kiinteys (cohesion)	Testattavuus (testability)
Kompleksisuus (complexity)	Ymmärrettävyys (understandability)
Rakenteellinen koko (size)	Ylläpidettävyys (maintainability)
Uudelleenkäyttö (reuse)	Tehokkuus (efficiency)
Modulaarisuus (modularity)	Kompleksisuus (complexity)
Suunnittelumallit (design patterns)	Luotettavuus (reliability)
Anti-suunnittelumallit (anti-patterns)	Siirrettävyys (portability)
	Yhteiskäytettävyys (interoperability)
	Koko (size)
	Laatu (quality)

Ohjelmistomittauksen vaikeudesta kertoo jotain se, että suorituskyky käytetyn muistin ja prosessoriajan määränä sekä ohjelmiston koko koodiriveinä tai kilotavuina ovat harvoja suureita, joita yleisesti osataan mitata eksaktisti. Yleisesti käytetään hyvin vähän numeerisia arvoja, jotka kertoisivat esimerkiksi, kuinka ylläpidettävä ohjelmisto on. Tämä johtuu siitä, että laatukriteerit ovat paitsi subjektiivisia ja tapauskohtaisia, myös epäformaaleja, tai ainakin esitetty epäformaalisti. Lisäksi kriteerit ovat usein ristiriitaisia. Osasta ominaisuuksa (mm. suorituskyky) ollaan yleisesti yhtä mieltä, vaikkakin kriteereiden painoarvot vaihtelevat ohjelmakohtaisesti (tietokanta- ja tietoliikenneohjelmien suorituskykyyn vaikuttaa itse ohjelmaa enemmän ulkoisten riippuvuuksien takana olevat seikat, kuten tietokannan hakuajat ja tietoliikenneyhteyksien nopeus).

Mikäli mitattava suure tiedetään, on vähimmäisvaatimuksena laskea suhteellisia arvoja, joiden mukaan ohjelmistoja voidaan vertailla keskenään. Mitta on mitattavan kohteen kuvaus tietyille, eksakteille arvoille, kuten kokonaisluvuille. Vaikka olisi vaikea määrittää, onko kiinteyden mitattu arvo 7 hyvä vai huono, se on selkeästi parempi kuin arvo 3 (oletettaen, että mitta-asteikko on kasvava, jolloin suuri koheesion arvo on suotavampi asia kuin pieni). Juuri ohjelmistojen mittaamisessa on ongelmana se, että mittauksen kohteen ominaisuudet eivät riipu ainoastaan sen rakenteesta, vaan myös semantiikasta. Esimerkiksi tyypillinen, hyvin suunniteltu aliohjelma tarkistaa saamansa parametrit, vaikkapa null-osoittimen aiheuttaman ajonaikaisen virheen välttämiseksi. Aliohjelma voi sisältää tätä varten ehtolauseen ("if muuttuja = null"). Pääasiassa kontrolli etenee ehtolauseessa oletusarvoista ns. päähaaraa ja kääntyy toiseen haaraan vain siinä tapauksessa, että parametri on virheellinen. Ohjelman suorituskyvyn mittaamista auttaisi, jos olisi tiedossa, mikä on kontrollin kulun todennäköisyys tietyssä ehtohaarassa. Se ei kuitenkaan riipu ohjelman syntaksista, vaan semantiikasta. Ja ohjelman semantiikka on asia, joka on hyvin huonosti formalisoitavissa.

Ohjelmistosta mitattavista suureista juuri edellämainitut suorituskyky ja koko ovat sellaisia, jotka on mahdollista mitata tarkasti vasta ohjelman ollessa valmis. Tällöin on usein työlästä tehdä ohjelmaan muutoksia, jotka parantaisivat mitattuja arvoja, mikäli se ylipäänsä enää on mahdollista tekemättä ohjelmaa uudelleen jopa alusta saakka. Valmiin ohjelman ominaisuudet riippuvat suurimmalta osin sen *arkkitehtuurista*, jonka kuvaus – toisin kuin ohjelmakoodi – on selvillä jo ohjelman suunnitteluvaiheessa. Aksioomaa *"laadukas prosessi tuottaa laadukkaita tuotteita"* voidaan tarkentaa: *"hyvä sisäinen rakenne johtaa hyvään ulkoiseen laatuun"* [Fen1991]. Olisi siten pystyttävä mittaamaan tai *ennakoimaan* ohjelman ominaisuuksia mahdollisimman varhaisessa vaiheessa, jotta mittaustulosten mahdollisesti aiheuttamat muutokset pystyttäisiin tekemään mahdollisimman vähäisellä kustannuksella [Rom1990]. Suunnitteluvaiheessa muutosten tekeminen on vielä edullista. *Ennakointi* (prediction) onkin tärkeä osa mittausta [Fen1991].

Ennakoivaan mittaamiseen kuuluu myös mittaustulosten tarkentuva todentaminen ohjelmistoprosessin aikana, mikä on ainoa tapa todentaa mittausstrategian kelvollisuus sekä suunnitellun arkkitehtuurin toteutuvuus toteutetussa ohjelmistossa. On selvää, että esimerkiksi suunnitteluvaiheessa mitatun kompleksisuuden olisi oltava pienempi tai yhtä suuri kuin lähdekoodista mitatun [HeS1990]. Mikäli näin ei ole, suunniteltuun arkkitehtuuriin on prosessin aikana tullut muutoksia, eikä ohjelmisto enää vastaa suunnitelmaa. Ohjelmistomittaamisen tarkoitus voi olla vain saada tietoa mitattavasta järjestelmästä, tai myös ennustaa ja tarvittaessa parantaa sen laatua [Rom1990]. Käänteistekniikan yhteydessä mittauksen syy saattaa olla ensiksimainittu, kun taas suunnitteluvaiheessa oletusarvoisesti jälkimmäinen.

Suunnittelumallit ovat kokemuksesta hyväksi havaittuja tapoja ratkaista tietty, yleinen suunnitteluongelma. Suunnittelumallit ovat dokumentoitua ohjelmoinnin ns. know-how-tietämystä. Vaikka suunnittelumallit liittyvät perinteisesti olio-ohjelmointiin, niitä voi soveltaa ohjelmoinnissa laajemminkin. Suunnittelumalleiksi monesti käsitetään ainoastaan "The Gang of Fourin" Design Patterns -kirjaan kerättyjä malleja mutta käsite itsessään on yleinen. Kokeneilla ohjelmistosuunnittelijoilla voi itselläänkin olla malleja, jotka he itse ovat keksineet tietyn ongelman yleisiksi ratkaisuiksi ja joita he itse käyttävät. Tällaisen, ohjelmoijien itsensä keksimä suunnittelumalli ei tosin yleensä ole dokumentoitu, paitsi ohjelmoijan omissa ajatuksissa. Design Patterns -kirjassa luetellut mallit on nimetty ytimekkäästi ja kuvaavasti, ja eräs mallien ideasta on löytää termejä, joilla ohjelmiston rakenteesta voitaisiin puhua karkeammalla tasolla kuin pelkkinä luokkina ja luokkien välisinä suhteina. Tässä on jo onnistuttukin. Mallien käyttöä ja niistä saatavaa hyötyä on kritisoitukin [Dod], mutta pääasiassa niistä on saatu positiivisia kokemuksia teollisuudessa [Bec1996, Sch1995].

Ohjelman rakennusosasina suunnittelumallit pitävät sisällään tietoa ohjelman toteutuksen ideasta ja uudelleenkäytettävyydestä. Niiden tunnistaminen olemassa olevasta ohjelmasta on sekin ohjelman ominaisuuksien mittaamista, koska tiettyyn suunnittelumalliin liittyvät ominaisuudet siirtyvät sitä käyttävään arkkitehtuuriin. Suunnittelumallien käyttö johtaa elegantteihin ja uudelleenkäytettäviin arkkitehtuureihin, ja näin ollen mallien tunnistaminen kertoo ohjelmiston arkkitehtuurin hyvästä laadusta. Mallin olemassaolosta voidaan päätellä ohjelman hyviä ominaisuuksia. Toisaalta, kullakin mallilla on myös haittavaikutuksensa. Lisäksi negatiivisen, ns. *anti-suunnittelumallin* (anti-pattern) löytyminen ohjelmistoarkkitehtuurista kertoo arkkitehtuurin huonosta laadusta ja tietyistä, anti-malliin liittyvistä ongelmista [CMG, Bro1998]. Anti-malleja on lueteltu kirjallisuudessa samoin kuin tavallisia, positiivisia suunnittelumallejakin.

4 Olioperustaiset ohjelmistometriikat

Seuraavassa luodaan yleiskatsaus 1990-luvulla kehitettyihin erityisesti oliopohjaisiin ohjelmistometriikoihin. Toisessa aliluvussa käsitellään olio-ohjelmien arkkitehtuurien mittausta ja suunnittelumallien tunnistamista.

4.1 Metriikoita ja historiaa

Proseduraalisten ohjelmien mittaamista on tutkittu lähes yhtä kauan kuin rakenteista, proseduraalista ohjelmointia on harrastettu, 1970-luvulta saakka. Olioperustaista ohjelmistomittausta on puolestaan tutkittu 1980-luvun lopulta ja erityisesti 1990-luvulla. Olioperustainen ja ns. perinteinen mittaaminen eivät kuitenkaan ole toisensa poissulkevat, vaan voidaan ajatella perinteisten menetelmien olevan osajoukko olio-ohjelmien mittaamisesta. Yleisimpiä (erityisesti arkkitehtuurimittaukseen soveltuvia) mittauksen kohteita ovat olleet kytkentä, kiinteys ja kompleksisuus sekä suorituskyky. Kompleksisuus implikoi usein – mutta ei välttämättä – myös suurta kokoa. Luetellut ominaisuudet ovat lähinnä ohjelmiston ulkoisia attribuutteja, ja niiden arvojen laskemiseksi on kehitetty alemman tason metriikoita. Joitakin vuosia sitten kritisoitiin, että tutkimuksissa käytetty testiaineisto on ollut riittämätöntä ja suppeaa ja peräänkuulutettiin teollisilla ohjelmistoilla todennettuja metriikoita [CoG1993]. Tästä on ajan kuluessa saatu positiivisia vastaesimerkkejä. Myös riittävien formaalien kuvausten puutetta on kritisoitu, ja monet tutkijat ovatkin kehittäneet omia formalismejaan. Tämänkin puutteen suhteen tilanne lienee pikkuhiljaa kehittymässä suotuisaan suuntaan. Suurin osa metriikoista mitataan ohjelmakoodista, mutta ohjelmistoarkkitehtuurien mittaaminen on herättänyt kiinnostusta 1990-luvun aikana.

Shyam Chidamber ja Chris Kemerer esittivät kehittämiään kiinteys- ja kytkentämetriikoita 1990-luvun alussa [ChK1991, ChK1994]. Heidän mittausjärjestelmäänsä kuuluu kuusi eri metriikkaa, jotka on muiden yleisimpien metriikoiden ohella lueteltu sivulla 18. Nämä metriikat ovat herättäneet viljalti kiinnostusta 1990-luvun loppupuolella: niihin on viitattu laajalti, niistä on keskusteltu, niitä on jopa kritisoitu, ja niitä ovat päivittäneet niin Chidamber ja Kemerer itse kuin muutkin tutkijat [Ban1998, Bas1996, BiS1998, BrC1994, Chi1998, Etz1999, Etz1998, Har1998, KaB1998, Mar1998]. Erityisesti on tarkasteltu *kiinteyden puute* -mittaa (Lack Of Cohesion in Methods, LCOM). On esitetty, että paras kiinteyden mitta olisi Lin ja Henryn vuonna 1993 päivittämä versio Chidamberin ja Kemererin alkuperäisestä LCOM-mitasta (on tosin huomattava, että väitteen esittäjiin kuuluu sama Li!) [Etz1998]. Lionel Briand, John Daly ja Jurgen Wust ovat hiljattain koonneet laajan ja kattavan yhteenvedon sekä kiinteyden että kytkentöjen mittaamisesta [Bri1998, Bri1999]. Kummassakin raportissa vertaillaan olemassa olevia metriikoita ja niiden luonnetta. Clark Archer ja Michael Stinson ovat koonneet olioperustaisista metriikoista yhteenvedon SEI:n (Software Engineering Institute) teknisessä raportissa [ArS1995]. Raportissa luetallaan useiden tutkijoiden 1990-luvun alkupuoliskon aikana kehittämiä mittausjärjestelmiä ja luodaan katsaus heidän tutkimuksiinsa.

Yleisesti on tultu siihen johtopäätökseen, että Chidamberin ja Kemererin ja niitä vastaavat metriikat toimivat ohjelmiston laadun indikaattoreina. Toinen usein esiintyvä johtopäätös on, että mittausjärjestelmään ei kannata lukea mukaan liian monta metriikkaa. Viidestä kymmeneen metriikkaa on ihanteellinen määrä, koska tällöin ohjelmaa tulee tarkasteltua riittävän monen metriikan näkökulmasta, mutta tarpeettoman monet lukuarvot eivät peitä yleiskuvaa [Mar1998]. Osassa tutkimuksista on käytetty hyvinkin montaa metriikkaa kerrallaan. Metriikoita on monia – on tiedettävä, mitä ominaisuuksia halutaan mitata, ja valittava rajallinen määrä metriikoita sen perusteella.

Pieni kytkentöjen määrä on toivottava asia, kuitenkin suhteutettuna siihen, että käytännössä kaikki ohjelman komponentit eivät koskaan ole täysin erillisiä, vaan niiden täytyy toimia yhteistyössä. On siis olemassa sekä toivottavia että epätoivottavia kytkentöjä. Eri tutkijoiden kehittämät mittausjärjestelmät ovat luokitelleet kytkennän lajeja hieman eri katsantokannoilta ja tarkkuuksilta, mutta muutoin hyvinkin yksimielisesti [Bri1999]. Kytkentöjä aiheuttavat metodikutsut ja yhteisten tietorakenteiden käyttäminen mitattavien entiteettien kesken. Tietorakenteisiin voidaan viitata joko suoraan tai välittää niitä metodien paluuarvoina ja parametreina. Yleensä metodikutsuihin perustuvia kytkentöjä pidetään toivottavina ja tietorakenteisiin perustuvia kytkentöjä epätoivottavina. Lisäksi kytkennän voi sanoa olevan joko suoraa tai epäsuoraa. Jos metodin parametriksi kirjoitetaan suoraan toinen metodikutsu, jonka arvoa siten käytetään ulomman metodin parametrina, puhutaan epäsuorasta kytkennästä; jos taas metodi sisältää viitteen toiseen olioon, kytkentä on suoraa. Yleensä suoraa kytkentää on pidetty toivottavampana, koska ohjelmassa olevia virheitä on tällöin helpompi jäljittää. Kytkennät ovat suunnattuja: kytkennän toisen osapuolen kannalta voidaan puhua ns. import- tai export-kytkennästä. Kahden tietyn entiteetin välisten kytkentöjen olisi aina oltava samansuuntaisia. Kytkentöjä voidaan mitata luokkien tai pakkausten tasolla.

On sanottu, että oliomaailmassa luokan käsite on korvannut perinteisen modulin [Bri1998]. Tässä yhteydessä on huomattava, että "modulilla" voidaan tarkoittaa joko luokan tapaista, tiedostoon tai komponenttiin rinnastettavaa yksikköä, tai nimiavaruuteen tai pakkaukseen verrattavaa kokonaisuutta. Tässä raportissa ja edellisessä viittauksessa "modulilla" tarkoitetaan ensiksimainittua käsitettä. Viimeeksimainittua, laajempaa kokonaisuutta luokka ei luonnollisesti korvaa, vaan pakkaukset ja nimiavaruudet ovat tyypillisesti oliokielissä käytössä luokkien ohella. Tällöin pakkaus tai nimiavaruus sisältää luokkia ja sisältämiinsä luokkiin liittyvän nimistön.

Olio-ohjelmoinnin kontekstissa kiinteys mittaa sitä, kuinka kiinteän kokonaisuuden luokan metodit ja kentät yhdessä muodostavat [Bri1998]. Suuri kiinteyden arvo on luokalta toivottava asia. Koska semanttista kiinteyttä on käytännössä mahdotonta mitata, kiinteyden mittaaminen perustuu kenttien ja metodien suhteiden ja yhteistoiminnan analysointiin. Työkalutyyppinen luokka, jonka palvelut ovat semanttisesti yhteenkuuluvia laskentapalveluita, mutta eivät käytä yhteisiä tietorakenteita, saava tästä johtuen tarpeettomankin pienen kiinteysarvon. Pienen luokasta lähtevien kytkentöjen määrän voisi kenties olettaa implikoivan entiteettien sisäistä kiinteyttä, ja päinvastoin, sillä mikäli luokka on kiinteä, sillä tuskin on tarvetta sitoa itseään kiinteästi toisiin luokkiin. Luokan itsensä palveluita voi käyttää suurikin joukko toisia luokkia, mikä ei ole ristiriidassa sen kanssa, että yksittäisestä luokasta lähtevien kytkentöjen määrän olisi oltava pieni.

Kompleksisuusmitta voi liittyvä algoritmiin tai olioarkkitehtuuriin. Algoritmien kompleksisuutta on perinteisesti mitattu vuokaavioista ja -verkoista tai Petri-verkoista. Periaatteessa tähän käyvät myös tila-automaatit. Yksi tunnetuimpia kompleksisuusmittoja on McCaben nimeä kantava syklomaattisen kompleksisuuden käsitteeseen perustuva mitta [Pre1997]. Arkkitehtuurin kompleksisuutta mitattaessa luokkien pieni kiinteys aiheuttaa suuren kytkentöjen määrän suhteessa luokkien määrään, mikä puolestaan saa aikaan suuren kompleksisuuden koko järjestelmässä. Samoin luokkien suuri kiinteys pitää yhdestä luokasta lähtevien kytkentöjen määrän pienenä, minkä ansiosta järjestelmän kompleksisuuskin pysyy pienempänä.

Seuraavassa luetellaan yleisimpiä metriikoita ja mittoja, ja kuvataan kukin niistä lyhyesti.

Nimi suomeksi	**Nimi englanniksi**	**Lyhenne**
Kiinteyden puute	Lack Of Cohesion in Methods	LCOM
Koodirivien lukumäärä	Lines Of Code	LOC
Operaattorien lukumäärä	Number of operators	
Perintäpuun syvyys	Depth ofInheritance Tree	DIT
Jälkeläisten lukumäärä	Number Of Children	NOC
McCaben kompleksisuusmitta		
Julkisten metodien määrä		
Metodien paino	Weighted Methods Per Class	WMC
Luokan kytkennät	Coupling Between Object Classes	CBO
Luokan vastausalue	Response for a Class	

Kiinteyden puute Kuuluisa perusmetriikka [ChK1994]. Idea on laskea sellaisten metodiklusterien määrä, jotka viittaavat samoihin kenttiin. Mitä vähemmän tällaisia erillisiä ryppäitä on, sitä suurempi on mitattavan kohteen kiinteys. Eri tutkijat ovat joko käsitelleet perintäsuhteen tuomat attribuutit ja metodit erikoistapauksina, tai eivät.

Koodirivien lukumäärä Koodirivejä laskettaessa on päätettävä, halutaanko laskea mukaan kommenttirivit ja/tai tyhjät rivit. Ohjelmoijan tyyli vaikuttaa tämän metriikan arvoihin, joten arvot eivät välttämättä ole keskenään vertailukelpoisia.

Operaattorien lukumäärä Yksi vaihtoehto edellä mainitulle metriikalle on laskea koodissa esiintyvien operaattorien määrä, joka ilmaisee sekä koodin kokoa että kompleksisuutta. Variaatioina voidaan laskea operaattorien ja operandien määrien suhde tai koodirivien ja operaattorien määrien suhde.

Perintäpuun syvyys Suuri arvo vähentää ohjelmiston ylläpidettävyyttä [ChK1994]. Metodien periytymisen ja mahdollisten (päällekkäistenkin) uudelleenmäärittelyjen seuraaminen pitkän hierarkian läpi on työlästä. Leveät mutta matalat puut ovat ylläpidettävyyden suhteen huomattavasti edullisempia (lisäksi tietyn, perityn rajapinnan ymmärtäminen auttaa ymmärtämään kaikkia periviä komponentteja). Osa suunnittelumalleista perustuu juuri syvän ja laajan perintäpuun hajoittamiseen esimerkiksi kahteen osaan (mikä toisaalta saattaa luoda epäsuotuisan voimakkaan riippuvuuden näiden kahden hierarkian välille).

Jälkeläisten lukumäärä Luokan **suorien** jälkeläisten lukumäärä [ChK1994]. Luokan kaikkien perillisten määrä saadaan laskettua summaamalla rekursiivisesti luokan suorien jälkeläisten NOC:t. Mittaa kompleksisuutta ja välillisesti kokoa. Vrt. perintäpuun syvyys.

McCaben kompleksisuusmitta Yksi yleisesti käytetyistä, perinteisistä kompleksisuusmetriikoista. Perustuu vuoverkon solmujen ja särmien suhteelle. Arkkitehtuurissa esiityvät kytkennät implikoivat osaltaan kompleksisuutta, samoin perintähierarkian syvyys ja julkisten metodien määrä.

Julkisten metodien määrä Rajapinnan koko. Implikoi kompleksisuutta ja myös kokoa. Variaatioina voidaan laajentaa näkökulmaa laskemalla luokan toteutettujen ja perittyjen metodien lukumäärien tai luokan itse määrittelemien ja perittyjen metodien lukumäärien suhde.

Metodien paino Luokan metodien kompleksisuuden summa [ChK1994].

Luokan kytkennät Niiden kytkentöjen määrä, joissa luokka on osallisena [ChK1994]. Laskettaessa koko järjestelmän CBO, on muistettava jättää huomiotta kytkentöjen duplikaatit, jotka tulisivat virheellisesti lasketuksi mukaan, jos järjestelmän CBO laskettaisiin vain laskemalla kaikkien järjestelmän luokkien CBO:t yhteen.

Luokan vastausalue Niiden metodien lukumäärä, joita luokan metodit kutsuvat, mukaanluettuna luokan kaikki omat metodit [ChK1994]. Arvoa ei lasketa rekursiivisesti transitiivisen sulkeuman summana, vaan mukaan lasketaan ainoastaan ensimmäiset metodikutsut, ja kukin metodi lasketaan summaan vain kerran, vaikka sitä kutsuttaisiin monestakin luokan metodista. Arvo mittaa luokan kompleksisuutta.

4.2 Perinteiset metriikat oliomaailmassa

Olioperustaisten ohjelmistojen mittaaminen on suhteellisen uutta, koska olioperustaisuus on paradigmana nuori. Voidaan miettiä, missä määrin perinteiset menetelmät purevat oliomaailmassa. Olioparadigma laajentaa proseduraalisen ohjelmoinnin käsitteitä *tiedon kapseloinnilla* ja *perinnällä*. Nämä uudet käsitteet vaativat kiinteyden ja kytkentöjen uudelleenmäärittelyä. Perintäsuhteita on jopa tarkasteltu erillisenä ilmiönä ("coupling, cohesion **and** inheritance") [Bri1998]. Aliluokan ja yliluokan välillä on kiinteä suhde, mikä lisää niiden kytkentöjä. Onko perintä siis epätoivottava asia? Toisaalta, aliluokka on yliluokkansa laajennos (IsA-suhde), jolloin sillä täytyykin olla saantioikeus kaikkiin yliluokkansa ominaisuuksiin. Tällöin perintä lisää kiinteyttä ja on siten toivottava asia. Syvät perintähierarkiat lisäävät ohjelman kompleksisuutta. Voitaisiin päätelllä, että matala mutta mahdollisesti leveä perintäpuu tuottaisi optimaalisimmat kiinteyden ja kytkentöjen arvot. On päätettävä, tarkastellaanko luokkia ja olioita perinnän muodostaman IsA-suhteen läpi, yksittäisinä kiinteinä entiteetteinä vai kerroksittaisina tietorakenteina, jotka koostuvat eri tasoilla tehdyistä määrittelyistä. Myös perinnän mahdollistama monimuotoisuus ja yliluokan metodien mahdolliset uudelleentoteutukset aliluokissa lisävät pohtimisen aiheita. Joidenkin oliokielien tukema *moniperintä* ei juurikaan lisänne edellä mainittuja, perinnän itsensä aiheuttamia kysymyksiä.

Stephen Schach tuo näiden kysymysten keskelle toisen näkökannan [Sch1996]. Oliokäsitteitä on mahdollista simuloida proseduraalisin menetelmin. Schachin mukaan olioiden ja luokkien välisen kiinteyden ja kytkentöjen mittaaminen ei olekaan niin monimutkaista kuin miltä näyttää, vaan perustuu vankasti proseduraalisiin mittareihin. Schachin mukaan ainoa erikseen tarkasteltava asia on perintä, jota ei voida ottaa mukaan mitattaessa kytkentöjä. Tällöin perinnän aiheuttamat kytkennät ovat toivottavia, kun taas muut kytkennät ja kiinteys mitataan perinteisten menetelmien mukaan. Luokkia voidaan käsitellä samoin keinoin kuin tavanomaisia moduuleja. Oliomaailmassa julkiset attribuutit ovat yhtä epätoivottuja kuin moduulien kesken jaetut globaalit muuttujat proseduraalisessa maailmassa, ja niin edelleen.

4.3 Olioperustaisen arkkitehtuurin mittaaminen

Kuten aiemmin mainittiin, mittaaminen ja muut laadunarviointimenetelmät tuottavat parhaan tuloksen, mikäli niitä sovelletaan ohjelmistoprosessin alkuvaiheessa. Tätä on tutkittu ja siitä on saatu hyviä tuloksia jo vuosikymmenen alussa, jolloin standardoituja oliopohjaisia arkkitehtuurin kuvauskieliä ei vielä juuri ollut [HeS1990, Rom1990]. Olioparadigma tukee ohjelmistoarkkitehtuurin kuvausta (puoli)formaalein menetelmin, mille suunnitteluvaiheessa tehty mittaaminen luonnollisesti perustuukin, koska ohjelmakoodia ei tällöin ole vielä saatavilla. Monet olioperustaisistakin metriikoista perustuvat kuitenkin ohjelmakoodin analysointiin eikä arkkitehtuurikuvauksiin. Suunnitteluvaiheen mittausta on tutkittu vähemmän. Erittäin vähän on tutkittu mittausta juuri UML-kuvauksista, koska UML on kuvaustekniikkana suhteellisen tuore. Se on kuitenkin nopeasti yleistymässä ja syrjäyttänyt OMT:n kuvaustekniikkana. Lisäksi UML:n ehdoton etu edeltäjiinsä nähden on monipuolisuus. Sille ei tällä hetkellä löydy varteenotettavaa kilpailijaa.

Mittaaminen nimenomaan arkkitehtuurikuvauksista asettaa omia rajoituksiaan. Kiinteyttä, kytkentöjä ja potentiaalista uudelleenkäytettävyyttä voidaan mitata siitä luontevasti. Arkkitehtuurikuvausten voisi ehkä ajatella tukevan kiinteyden ja kytkentöjen mittaamista, kun taas ohjelmakoodin voisi katsoa tukevan suorituskyvyn mittaamista. Arkkitehtuurikuvausten korkea abstraktiotaso hankaloittaa suorituskyvyn mittaamista. UML:iin kuuluvista tila-automaateista voidaan periaatteessa mitata algoritmin tehokkuutta tai kompleksisuutta samaan tapaan kuin vuokaaviosta tai vuoverkosta. Toisaalta yhden solmun tila- tai aikavaatimus on hankalampi määrittää tila-automaatista kuin vuokaaviosta tai -verkosta, koska UML on korkeamman tason kuvaustapa, eivätkä algoritmin kaikki yksityiskohdat ole välttämättä selvillä vielä suunnitteluvaiheessa. Ohjelmakoodi on aina tietyllä tarkkuustasolla, kun taas rakennekuvaus voi olla hyvinkin ylimalkainen (esimerkiksi “asiakas-palvelin”). Mitä tarkemmalla tasolla kuvaus on tehty, sitä tarkempia voi myös mittaustulosten olettaa olevan. Jos UML-kuvauksessa olisi mukana myös metodien väliset kutsuriippuvuudet, se helpottaisi esimerkiksi suorituskyvyn tarkempaa mittaamista.

UML:n luokkakaavio kuvaa ohjelman staattisen rakenteen. Gail Murphyn ja Ekaterina Saenkon mukaan ohjelmiston tilavaatimuksen mittaaminen luokkakaavion perusteella on mahdollista [MuS1998]. Heidän tekemänsä tutkimus perustui ohjelmakoodin analysointiin ja ohjelman muistihallinnasta saatuun metadataan ja siihen, että ohjelman suoritusaikainen muistinvaraus perustuu olioiden luomiselle. Tekniikan voikin päätellä soveltuvan parhaiten puhtaisiin oliokieliin, joissa kaikki data on olioina ja varsinaisia globaaleja tietorakenteita ei ole. Murphy ja Saenko arvioivat dynaamisesti luotujen olioiden määrää tietyn luokan ilmentymien lukumäärän perusteella. Tätä luokkaa kutsutaan kantaluokaksi ja käyttäjä antaa sen ilmentymien lukumäärän. Tarkastellaan graafisen käyttöliittymän esimerkkiä, jossa kantaluokkana käytetyssä ikkuna-luokassa on keskimäärin 15 graafista komponenttia ohjelmalle annetaan tieto, että ikkunoita luodaan keskimäärin kolme ja pahimmassa tapauksessa viisi kappaletta, voidaan ohjelman odottaa luovan keskimäärin 45 oliota. Työkalu laskee muistin tarpeen keskiarvon lisäksi pahimman tapauksen.

Murphyn ja Saenkon teoria vaikuttaisi lupaavalta. Toisaalta, suoritusaikaiseen muistinkulutukseen vaikuttavat myös (mahdollisesti rekursiivisestikin tehdyt) metodikutsut aktivaatiotietuepinoineen. Mittaustyökalun vuorovaikutus käyttäjän kanssa on tässäkin tapauksessa tärkeää, jotta olisi mahdollista estimoida luotujen olioiden määrää. Lähteen tutkimuksessa jouduttiin metadatan saamiseksi muun muassa uudelleenmäärittelemään luokkien konstruktoreja ja destruktoreja, mikä ei käytännössä ja pitkällä tähtäimellä ole enää mielekästä – tähän olisi saatava parempi tukijärjestelmä. Toisaalta Murphy ja Saenko muodostivat luokkakaavionsa ohjelmakoodin pohjalta. Jos mittauksen perustana käytettäsiin suoraan UML- tai vastaavia arkkitehtuurikuvauksia, se yksinkertaistaisi mittausprosessia ainakin jonkin verran.

Arkkitehtuurin mittaamisen suunnitteluvaiheessa suunnittelumallien kautta, ja ylipäätään suunnittelumallien tunnistamisen arkkitehtuurista suunnitteluvaiheessa voidaan sanoa olevan "forward engineeringiä reverse engineeringin keinoin". Tehdyissä tutkimuksissa on havaittavissa hämmästyttävän selkeä kahtiajako käänteistekniikkaan (reverse engineering) kuuluvan suunnittelumallien analyysin ja eteenpäin kulkevaan ohjelmistotuotantoon (forward engineering) lukeutuvan tavanomaisen mittaamisen välillä. Suunnittelumalleihin liittyvä mittaaminen ja suunnittelumallien tunnistaminen on vielä tuore tutkimuksen ala. Itse suunnittelumallitkin ovat suhteellisen nuori käsite verrattuna proseduraalisen ohjelmoinnin ja ohjelmistojen mittaamisen pitkään historiaan.

Suunnittelumallien tunnistaminen on yllättävän vähän tutkittu ja yllättävän vaativa ongelma [Mis1997, Bro1996, Flo1997, Shu1996, KrP1996]. Tarvitaan ehtoja, jotka ohjelmistorakenteen on täytettävä toteuttaakseen tietyn suunnittelumallin, ja ehdot on kuvattava riittävän formaalilla tavalla. Mallit on usein ilmaistu löyhästi. Osa ehdoista on selviä, mutta osa on tulkittava valinnaisiksi. Ongelmaksi muodostuu myös ohjelmointikielen rakenteiden hyödyntäminen suunnittelumallien toteutuksessa. Kielen tukemien erikoispiirteiden (esim. C++:n template-mekanismi) avulla malleja voi kenties toteuttaa hyvinkin paljon tehokkaammin kuin pelkkiin olioparadigman peruselementteihin tukeutuen. Tämä tekee kuitenkin mallien toteutuksista entistä vaikeampia tunnistaa. Johonkin raja on vedetttävä, ja toteutuksen on täytettävä tietyt rakennekriteerit. Lisäksi osa malleista on helpompi mallintaa säännöiksi, kun taas osa on hyvinkin vaikeita tunnistaa. Kuten edellisessä luvussa mainittiin, ohjelmiston ominaisuudet eivät riipu yksinomaan sen rakenteesta, vaan suurelta osin sen semantiikasta. Tämä on ongelmallista erityisesti suunnittelumallien tunnistamisessa: arkkitehtuurista saattaa löytyä suunnittelumallia muistuttava rakenne, joka ei kuitenkaan ole mallin oikea ilmentymä, eikä sen ole tarkoituskaan sitä olla.

Pauli Misikankaan Helsingin yliopistossa tekemä pro gradu -tutkielma yhdistää suunnittelumallien tunnistamiseen tekoälyn etsintätekniikat [Mis1997]. Tutkielmassa suunnittelumallit kuvataan tietämyskannassa Prolog-faktoina. Mallin tunnistaminen esitetään ehdontäyttämisongelmana (constraint satisfaction problem), johon sovelletaan onnistuneesti epäkelpojen vaihtoehtojen poisrajaamistekniikkaa. Misikankaan tekemä Java^{TM}-kielinen ohjelma suoriutuukin etsinnästä nopeammin kuin Prolog-tulkki ja muut tunnetuimmat etsintätekniikat.

Jotta mittausta pystyttäisiin suorittamaan taloudellisesti ja sen tuloksia analysoimaan tehokkaasti, mittaamisen on oltava automatisoitavissa. Automaattiset työkalut ovat olennainen osa metriikoiden evaluointia käytännössä. Mielipiteet ovat olleet melko yhtenäisiä siinä, että mittaamista ei usenkaan ole mielekästä tehdä täysin koneellisesti, vaan vuorovaikutuksessa käyttäjän kanssa [Mis1997, Rom1990]. Erityisesti suunnittelumallien tunnistamisessa tarvitaan käyttäjän vahvistus siitä, onko löydetty rakenne todella mallin oikea ilmentymä vai muistuttaako se vain mallin rakennetta siinä määrin, että työkalu on sen arkkitehtuurista tunnistanut. Toisena esimerkkinä, Murphyn ja Saenkon metriikkaidea vaatii käyttäjän antamaa tietoa siitä, mitä luokista käytetään kantaluokkana, jonka perusteella muiden luokkien ilmentymien lukumäärää arvioidaan.

Mittaustutkimuksen yhteydessä on tyypillisesti tehty mittaustyökalu. On olemassa kaupallisiakin mittausohjelmistoja, kuten C++-pohjainen QMOOD++ [BaD1997]. Se tukee viittä luokkien välisiin suhteisiin perustuvaa metriikkaa (muun muassa DIT, NOC) ja yhtätoista luokan sisäistä metriikkaa (muun muassa luokan koko tavuina sekä luokan attribuuttien ja metodien määrä) [Har1998]. Usean Euroopan maan välisen FAMOOS-tutkimushankkeen yhteydessä on tehty useita metriikka- ja visualisointityökaluja [Riv1999, FAMOOS]. Niiden käyttäjän on mahdollista esimerkiksi valita laatujärjestelmänsä, jonka mukaan metriikkatyökalu osaa laskea ohjelmiston laadulle kokonaisarvon. Koska metriikoita ja laatujärjestelmiä on lukuisia ja niitä kehitetään koko ajan lisää, työkalujen on tärkeää olla laajennettavissa tai ainakin konfiguroitavissa vastaavalla tavalla.

Metriikkaohjelmistot ja -tutkimukset ovat suurimmaksi osaksi perustuneet ohjelmakoodin analysointiin. Kiinnostus arkkitehtuurikuvausten mittaamiseen on kuitenkin lisääntymässä. Ohjelmakoodipohjaiset metriikat muuntavat usein koodin metadatakuvaukseksi, josta varsinaiset mittaukset tehdään. Misikankaan suunnittelumallien tunnistaja muuttaa Java^{TM}-koodin Prolog-faktoiksi. FAMOOS-työkalut käyttävät ascii-pohjaista Famix-kuvausjärjestelmää. Pohja koodia korkeamman tason analysoinnille on siis olemassa. Edullista olisi integroida tai muuten yhdistää mittaustyökalu koodia generoivaan CASE-välineeseen. Tällöin mittaaminen (joko koodiin, arkkitehtuurikuvaukseen tai kumpaankin pohjautuen) olisi jo teknisesti lähempänä käytännön ohjelmistokehitystä ja ohjelmistotyöntekijää.

Muun muassa Murphyn ja Saenkon tutkimus käsitteli arkkitehtuurikuvauksen mittaamista, vieläpä uudelta, ohjelman tilavaatimuksen arvioinnin kannalta. Kiinteyden, kytkentöjen ja kompleksisuuden mittaamista arkkitehtuurista on tutkittu useammilla tahoilla. Martin Hitz ja Karin Neuhold ovat esittäneet ajatuksiaan rakenteisten, korkean tason suunnitteluformalismien mittaamisesta [HiN1998]. Hekin painottivat käytetystä ohjelmointikielestä riippumatonta ohjelmistojen mittausta. The Metrics of Object Oriented Technology -tutkimusryhmä on samoilla jäljillä [MOT]. Projekti on tuore, eikä ryhmä ole ehtinyt julkaista tuloksia vielä raportin valmistumisen aikaan. Arkkitehtuurin mittausta tutkineet ryhmät eivät välttämättä ole ottaneet kantaa arkkitehtuurin kuvaukseen tai olioperustaisuuteen. Michele Marchesi on kuitenkin tutkinut arkkitehtuurimetriikoita UML-kuvauksista näennäisen menestyksekkäästi [Mar1998]. Marchesin tutkimustyö on vielä kesken, ja hänen metriikoitaan todennetaan vielä tutkimusorganisaation ulkopuolisella testiaineistolla.

5 Avoimia kysymyksiä

Kun ohjelmistosta tunnistetaan suunnittelumalli, on päästy vasta metriikan tasolle. Mitä voidaan päätellä siitä, että ohjelmassa on tietyn mallin ilmentymä? Mitä se oikeastaan tarkoittaa? Mallit voivat lisäksi olla päällekkäisiä, toisin sanoen sama luokka voi täyttää roolin useammassa kuin yhdessä mallissa. Mallien vaikutukset saattavat olla yhteydessä toisiinsa. Vaikutus tiettyyn ominaisuuteen (vaikkapa suorituskykyyn) voi kumoutua tai kumuloitua. Jonkin verran on tutkittu yksittäisten mallien seurauksia – jo "The Gang of Fourin" kirjassa on analysoitu mallien soveltamisen vaikutuksia ohjelmistoon sekä mallien hyviä ja huonoja puolia – mutta niiden yhteisvaikutusta on tutkittu verrattain vähän.

Mielenkiintoista olisi tutkia suunnittelumallien esiintymisen korrelaatiota perinteisten mittausten tuloksiin. Tarkastellaan ohjelmaa, jossa on hyödynnetty suunnittelumalleja tiiviisti siten, että yhdellä luokalla on tyypillisesti rooli useassakin eri mallin ilmentymässä. Onko ohjelma tällöin kompleksinen, mahdollisesta pienestä koostaan huolimatta? Miten useat roolit vaikuttavat luokan kiinteyteen? Mitä voidaan päätellä suunnittelumallin määrän suhteesta ohjelman kokoon tai malleihin sidottujen luokkien suhteesta sellaisiin, joilla ei ole roolia missään mallissa? Miten nämä arvot suhtautuisivat esimerkiksi LCOM-mittaan? Ja tekeekö mallien kovin tiivis soveltaminen ohjelmasta vaikeammin vai helpommin ylläpidettävän?

On valitettu, että ohjelmistojen ja niiden ominaisuuksien kuvaamiseen ei ole ollut riittäviä formalismeja, joita ohjelmistojen mittaamisessa tarvittaisiin. Ylipäätään formalismit ovat edellytys mittaamiselle. UML on osaratkaisu tähän. Sen parhaat puolet ovat määritelty standardi, monipuolisuus ja kieliriippumattomuus. UML on kuvauskielenä tullut yhä suositummaksi. On jo olemassa useita UML:ia tukevia kaupallisia CASE-välineitä. Suunnittelumallien toiminta ja rakenne on kuvattu korkealla abtraktiotasolla, kuten UML ("design level – pattern level"). Voisi ajatella, että yhteinen abstraktiotaso helpottaisi mallien tunnistamista UML:n tapaan kuvatusta arkkitehtuurista. Suurin ongelma on, että UML-kuvauksille ei ole olemassa formaalia, standardoitua, esimerkiksi tekstuaalista tallennusformaattia. UML-pohjaiset CASE-työkalut käyttävät kukin omaa sisäistä formaattiaan. Erityisesti monen käyttäjän ryhmätyöhän tarkoitetut CASE-välineet käyttävät kuvauksien tallettamiseen tietokantaa. Vaikka UML on standardoitu, sitä käytetään eri tavoin eri tahoilla. Eri CASE-välineillä on omia UML-laajennoksiaan, esimerkiksi tapa kuvata reaaliaikajärjestelmien ominaisuuksia. UML-kuvausten semantiikka ei ole lainkaan yhtä standardoitu kuin syntaksi. Jotkut standardinkin mukaan tehdyistä UML-kaavioista ovat sisällöltään epätarkkoja. Epämääräisistä dokumenteista on mahdotonta saada aikaan tarkkoja mittaustuloksia. Osaparannus semantiikan epätarkkuuteen on UML:n kylkiäiseksi kehitetty Object Constraint Language (OCL). Sen avulla voidaan luokille ja olioille esittää tarkempia ehtoja ja määrittelyjä kuin pelkällä UML:lla.

OCL:ia voi käyttää kuvaamaan ohjelmiston reaaliaikaisuuteen, synkronointiin ja rinnakkaisuuteen liittyviä vaatimuksia. Niin kutsuttu *Real-Time UML* on niiden ilmaisemiseen vielä luontevampi vaihtoehto, sillä se on kehitetty tukemaan nimenomaan tukee edellä lueteltuja ominaisuuksia [Dou1998]. Tutkimuksissa, joissa oli käytetty mittauksen lähtökohtana UML-kuvauksia, ei kuitenkaan ole viitattu ohjelmistojen reaaliaikaisuuteen, synkronointiin ja rinnakkaisuuteen liittyviin vaatimuksiin. Se on puute, sillä erityisesti suorituskyvyn mittaaminen kiinnostaa ohjelmiston kehittäjiä, jotka ovat tekemisissä reaaliaikaisten ja rinnakkaisten ohjelmistojen kanssa. Mikäli ohjelmistoarkkitehtuurien kuvauskielenä käytettäisiin reaaliaika-UML:ia, olisi mahdollista mitata esimerkiksi sitä, onko tietty tehtävä edes teoreettisesti mahdollista suorittaa tietyssä ajassa tai saadaanko se suoritettua tässä ajassa pahimmassakin tapauksessa. Suurin osa kaikista tuotetuista tietokoneohjelmista ovat reaaliaikaisia tai sisältävät ylipäätään aikavaatimuksia. Näin ollen reaaliaika-UML:n käyttö mittaamisessa olisi vahvasti perusteltua.

Luokkakaavio on yleisimmin käytetty olioperustainen tapa kuvata ohjelman rakennetta. Marchesin metriikkatyökalu ei käytä UML:n niitä kaavioita, jotka kuvaavat ohjelman dynaamista toimintaa. Myös Murphyn ja Saenkon tekniikka käyttää UML:sta vain luokkakaaviota. UML:iin sisältyy useita muitakin kaaviotyyppejä kuin luokkakaavio. Kiintoisa ongelma on, missä määrin ohjelman kuvauksia voitaisiin mitata myös muista kaavioista. Ainakin tila-automaatit soveltuvat suorituskyvyn mittaamiseen, vaikkei niitä ole siihen alun perin tarkoitettukaan. Olioyhteistyökaaviosta (object interaction graph) voitaisiin mitata oliotason kiinteyttä ja kytkentöjä sekä jossain määrin myös suorituskykyä. Viestivälityskaavio (message sequence chart) kuvaa olioiden suoritusaikaista viestinvälitystä tietyllä ohjelman suorituskerralla. Sillä voitaisiin kuvata esimerkiksi suorituskyvyn kannalta pahin tai keskimääräinen tapaus.

UML:lle ei näillä näkymin ole varteenotettavaa kilpailijaa olio-ohjelmien suunnittelutasoisena kuvaustekniikkana. Sen formaaleille esitystavoille on kuitenkin olemassa vaihtoehtoja. Marchesin metriikkatyökalu analysoi eräällä CASE-välineellä tehtyjä UML-kuvauksia, jotka käännetään prolog-faktoiksi [Mar1998]. Myös Misikankaan toteuttama suunnittelumalleja tunnistava työkalu kääntää Java^{TM}-koodin Prologiksi [Mis1997]. FAMOOS-työkalut käyttävät Famix-kuvaustekniikkaa, josta tutkijoilla on ollut hyviä kokemuksia [FAMOOS]. Prolog-sääntöjä voidaan evaluoida Prolog-tulkin avulla, minkä johdosta prologilla kuvattuja metriikkasääntöjä on mahdollista evaluoida ilman erillistä metriikkalaskinta (kunhan mittausaineisto on ensin käännetty Prolog-faktoiksi).

On ylipäätään pohtimisen arvoista miettiä, missä määrin korkean abstraktiotason kuvaukset eroavat mittausaineistona ohjelmakoodista. UML:n kaltaisten kuvausten etu on luonnollisesti kieliriippumattomuus. Ohjelmakoodista mittaamiseen vaikuttavat kielen piirteet, kirjastot ja rakenteet. Kuinka helppoa on tunnistaa kirjastoituun säiliöluokkaan perustuva kytkentä? Lisäksi kytkennät ja assosiaatiot ovat jo valmiiksi korkeamman tason käsitteitä kuin kentät ja luokkajäsenet. Kytkentöjen laatua on kuitenkin vaikeampi päätellä pelkästä arkkitehtuurikuvauksesta, ellei käytössä ole esimerkiksi "uses"-tyyppistä assosiaatiota. Kytkennän suuntakaan ei välttämättä näy kaaviosta, mutta se tieto on mahdollista lisätä assosiaatioon. Korkean tason kuvaukset olisi siis tehtävä riittävän tarkoiksi ja informatiivisiksi, jotta niistä olisi laskettavissa riittävän tarkkoja mittausarvoja. Tämä pätee eritoten kiinteyteen. Ainakaan UML ei sisällä kaaviotyyppiä, joka olisi *tarkoitettu* ohjelman suorituskyvyn tarkastelua varten. Voitaisiin ajatella, että korkean tason kuvaukset tukevat kytkentöjen ja korkean tason kompleksisuuden mittausta, kun taas ohjelmakoodi tukee suorituskyvyn ja algoritmisen tason kompleksisuuden mittaamista, mutta hieman monimutkaistaa kytkentöjen mittaamista.

6 Yhteenveto

Mittaaminen on tärkeää ohjelmistoprosessille. Mittaaminen on tapa varmistaa ja kehittää ohjelmiston laatua. Mitä varhaisemmassa vaiheessa mittaamista tehdään, sitä laajemmin ja sitä vähemmällä kustannuksella mittaustulokset voivat vaikuttaa ohjelmiston kehitykseen. Suurin hyöty mittaamisesta saadaan, jos se tehdään jo suunnitteluvaiheessa, ohjelmiston arkkitehtuurin kuvauksesta. Kuvauksen on kuitenkin oltava riittävän formaalisti esitetty, jotta mittaaminen olisi mahdollista. Lisäksi mittaamista on hyvä suorittaa koko ohjelmistoprosessin ajan.

Olioparadigma on tuonut tavan kuvata ohjelmiston toimintaa ja rakennetta jo suunnitteluvaiheessa. UML on monipuolinen, puoliformaali ohjelmiston kuvaustekniikka, jolla tehtyjä dokumentteja on mahdollista käyttää mittauksen kohteena. Luokkakaavioiden perusteella voidaan arvioida tulevan ohjelmiston kokoa ja tila-automaateista on mahdollista laskea algoritmien kompleksisuusmittoja. UML:n luonne korkean tason kuvauskielenä vaikuttaa kuitenkin mittauksen tarkkuuteen ja vaatii ohjelmiston kuvausta riittävän yksityiskohtaisesti.

Perinteiset mittausmenetelmät sopivat ainakin suurelta osin myös olioperustaisten ohjelmistojen mittaamiseen. Olioparadigman tukema luokkien perintä aiheuttaa kuitenkin osittaisia uudelleemäärittelyjä kiinteyteen ja kytkentään.

Tuore näkökulma olioperustaiseen ohjelmistomittaamiseen on suunnittelumallien tunnistaminen arkkitehtuurikuvauksesta ja toteutettavan ohjelmiston ominaisuuksien ennakoiminen löydettyjen mallien ominaisuuksien perusteella. Löydetyt mallit kertovat ohjelmiston hyvästä suunnittelusta, minkä lisäksi kullakin mallilla on tyypillisiä positiivisia ja negatiivisia ominaisuuksia. Tavanomaisten suunnittelumallien lisäksi arkkitehtuurista voidaan etsiä ns. anti-suunnittelumalleja, jotka ilmaisevat huonoja suunnitteluratkaisuja ja mahdollisia tulevia ylläpidettävyysongelmia.

Kuten perinteinen mittaus, myös suunnittelumallien, anti-suunnittelumallien ja vastaavien rakenteiden tunnistaminen on automatisoitavissa. Kuitenkin erityisesti malleihin perustuva mittaustyökalu hyötyy vuorovaikutuksesta käyttäjän kanssa.

Viitteet

[ArS1995] Clarc Archer, Michael Stinson: *Object-Oriented Software Measures*, Technical Report SMU/SEI-95-TR-002 ESC-TR-95-002 Software Endineering Institute, 1995.

[BaD1997] Jagdish Bansiya, Carl Davis: *Automated Metrics and Object-Oriented Development: Using QMOOD++ for Object-Oriented metrics*, Dr. Dobb's Journal, December 1997.
http//www.joopmag.com

[Ban1998] Jagdish Bansiya, Letha Etzkorn, Carl Davis, Wei Li: *A Class Cohesion Metric For Object-Oriented Designs*, JOOP, January 1999, 47-52.

[Bas1996] Victor Basili, Lionel C. Briand, Walcello L. Melo: *A Validation of Object-Oriented Design Metrics as Quality Indicators*, IEEE Transactions on Software Engineering, vol. 22, no. 10, Oct. 1996, 751-761.

[Bec1996] Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch, John Vlissides: *Industrial Experience with Design Patterns*, Proc. 18th International Conference on Software Engineering, IEEE CS Press, 1996, 103-114.

[Bei1995] Boris Beizer: *Black-Box Testing*, Wiley, 1995.

[BiS1998] Aaron B. Binkley, Stephen R. Schach: *Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures*, proc. 20 th Inernational Conference on Software Engineering, 1998, IEEE CS Press, 1998, 452-455.

[Bri1998] Lionel C. Briand, John W. Daly, Jurgen Wust: *A Unified Framework for Cohesion Measurement in Object-Oriented Systems*, Empirical Software Engineering, no. 3, 65-117, 1998.

[Bri1999] Lionel C. Briand, John W. Daly, Jurgen Wust: *A Unified Framework for Coupling Measurement in Object-Oriented Systems*, IEEE Transactions on Software Engineering, vol. 25, no. 1, Jan./Feb. 1999, 91-121.

[BrC1994] Fernando Brito e Abreu, Rogerio Carapuca: *Candidate Mertics for Object-Oriented Software within a Taxonomy Framework*, J. Systems and Software, 26, 87-96, 1994.

[Bro1996] Kyle Brown: *Design reverse-engineering and automated design pattern detection in Smalltalk*
http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm

[Bro1998] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. III McCormick, Thomas J. Mowbray: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, 1998.

[ChK1991] Shyam R. Chidamber, Chris F. Kemerer: *Towards a Metrics Suite for Object-Oriented Design*, proc. Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) 1991, ACM Sigplan notices Nov. 1991, 197-211.

[ChK1994] Shyam R. Chidamber, Chris F. Kemerer: *A Metrics Suite for Object-Oriented Design*, IEEE Transactions on Software Engineering, vol. 20, no. 6, June 1994, 476-493.

[Chi1998] Shyam R. Chidamber, David P. Darcy, Chris F. Kemerer: *Managerial Use of Metrics for Object-Oriented Software: An Explanatory Analysis*, IEEE Transactions on Software Engineering, vol. 24, no. 8, Aug. 1994, 629-639.

[CMG] Component Management Groupin anti-pattern -sivu
http://160.79.202.73/Resource/AntiPatterns/index.html

[CoG1993] Richard E. Courtney, David A. Gustafson: *Shotgun Correlations in Software Measures*, Software Engineering Journal, January 1993, 5-13.

[Dod] Mahesh Dodani: Rules Are for Fools, Patterns Are for Cool Fools, kolumni Joop Onlinessa
http://www.joopmag.com/html/from_pages/column.shtml

[Dou1998] Bruce Powel Douglass *Real-Time UML – Developing Efficient Objects for Embedded Systems*, Addison-Wesley, 1981.

[Etz1999] Letha Etzkorn, Jagdish Bansiya, Carl Davis: *Design and Code Complexity Metrics for OO Classes*, JOOP, March/April 1999, 35-40.

[Etz1998] Letha Etzkorn, Carl Davis, Wei Li: *A Practical Look at the Lack of Cohesion in Methods Metric*, JOOP, September 1998, 27-34.

[FAMOOS] FAMOOS-projekti http://www.iam.unibe.ch/˜ famoos

[Fen1991] Norman Fenton: *Software Metrics: A Rigorous Approach*, Chapman-Hall, London, 1991.

[FeP1996] Norman Fenton, Shari Lawrence Pfleeger: *Software Metrics: A Rigorous and Practical Approach*, International Thompson Computer Press, London, 1996.

[Flo1997] Gert Florijn, Marco Meijers, Pieter van Winsen: *Tool Support for Object-Oriented Patterns*, Proc. 11th European Conference of Object-Oriented Programming (ECOOP), 1997, 472-495.

[Gam1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1995.

[Har1998] Rachel Harrison, Steve J. Counsell, Reuben V. Nithi: *An Evaluation of the MOOD Set of Object-Oriented Software Metrics*, IEEE Transactions on Software Engineering, vol. 24, no. 6, June 1998, 491-496.

[HeS1990] Sail Henry, Calvin Selig: *Predicting Source-Code Complexity at the Design Stage*, IEEE Software, vol. 7, no. 2, March 1990, 36-43.

[HiN1998] Martin Hitz, Karin Neuhold: *A Framework for Product Analysis*, OOPSLA 1998 Workshop on Model Engineering, Methods and Tools Integration with CDIF, 1998.
http://www.metamodel.com/oopsla98-cdif-workshop/hitz.txt

[Hum1990] Watts S. Humphrey: *Managing the Software Process*, Addison-Wesley, 1990.

[Hum1995] Watts S. Humphrey: *A Discipline for Software Engineering*, Addison-Wesley, 1995.

[ISO] ISO9001: *Quality Systems - Model for Quality Assurance in Design, Development, Production, Installation and Servicing.*

[KaB1998] Rick Kazman, Marcus Burth: *Assessing Architectural Complexity*, Proc. 2nd Euromicro Conference on Software Maintenance and Reengineering, 1998, IEEE CS Press, 1998, 104-112.

[KrP1996] Christian Krämer, Lutz Prechelt: *Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software*, Proc. Working Conference on Reverse Engineering, 1996, IEEE CS Press, 208-215.

[Mar1998] Michele Marchesi: *OOA Metrics for the Unified Modeling Language*, Proc. 2nd Euromicro Conference on Software Maintenance and Reengineering, 1998, IEEE CS Press, 1998, 67-73.

[Men1994] Daniel A. Menascé, Virgilio A. F. Almeida, Larry W. Dowdy: *Capacity Planning and Performance Modeling*, Prentice-Hall, 1994.

[Men1996] Tim Menzies: *Patterns in Inference Strategies: Lessons and Potential Pitfalls for Reuse Design Patterns*, 1996.
http://www.cse.unsw.EDU.AU/˜ timm/pub/docs/96probspatt.ps

[Mis1997] Pauli Misikangas: *Suunnittelumallien automaattinen tunnistaminen olio-ohjelmista*, pro gradu -tutkielma, Helsingin yliopisto, tietojenkäsittelytieteen laitos, C-1998-1, 1997.

[MOT] The Metrics for Object Tehcnology
http://dec.bournemouth.ac.uk/staff/ckirsopp/mot.htm

[MuS1998] Gail C. Murphy, Ekaterina Saenko: *Predicting Memory Use from Class Diagram using Dynamic Information*, proc. First International Workshop on Software and Performance, 1998, ACM Order Number 488983, 145-151.

[Mye1979] Glenford J. Myers: *The Art of Software Testing*, Wiley, 1979.

[Pau1993] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, Charles V. Weber: *Capability Maturity Model, Version 1.1*, IEEE Software, vol. 10, no. 4, July 1993, 18-27.

[Pet1981] James L. Peterson: *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[Pot1996] Ben Potter, Jane Sinclair, David Till: *An Introduction to Formal Specification and Z*, 2nd edition, Prentice-Hall, 1996.

[Pre1997] Roger S. Pressman: *Software Engineering - A Practitioner's Approach*, 4th ed. McGraw-Hill, 1997.

[Riv1999] Claudio Riva, Michael Przybilski, Kai Koskimies: *An Integrated Environment for Reengineering Object-Oriented Software*, submitted to: 4th European Conference on Software Maintenance and Reenginering (CSMR), 2000.

[Rom1990] Dieter Rombach: *Design Measurement: Some Lessons Learned*, IEEE Software, vol. 7, no. 2, March 1990, 17-24.

[Rus1991] Glen W. Russell: *Experience with Inspection in Ultralarge-Scale Developments*, IEEE Software, vol. 8, no. 1, Jan. 1991, 25-31.

[SaC1981] Charles H. Sauer, K. Mani Chandy: *Computer Systems Performance Modeling*, Prentice-Hall, 1981.

[Sch1996] Stephen R. Schach: *The Cohesion and Coupling of Objects*, JOOP, January 1996, 48-50.

[Sch1995] Douglas C. Schmidt: *Using Design Patterns to Develop Reusable Object-Oriented Communication Software*, Comm. ACM, vol. 38, no. 10, Oct. 1995, 65-74.

[Shu1996] Forrest Shull, Wacelio L. Melo, Victor R. Basili: *An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems*, Tech. Rep. CS-TR-3597, University of Maryland, Computer Science Department, 1996.

[Smi1990] Connie Smith: *Performance Engineering of Software Systems*, Addison-Wesley, 1990.

[UML] UMLanguage Resource Center: Unified Modeling Language, Standard Software Notation
http://www.rational.com/uml/index.jtmpl